COMP 621 Program Analysis and Transformations
Assignment #1

Profiling and Traditional Flow Analysis

Due: Thursday Oct 6th, beginning of class

Milestone update (via e-mail) due on Monday, Sept 26

Overview:

The purpose of this assignment is two-fold. The first purpose is to participate in the collection or creation of benchmarks that will be used later in the course for testing our optimization, parallelization, and analysis methods. The second purpose is to give you some experience with stating and solving dataflow problems.

Question 1: *Benchmark Programs*

Benchmark programs form an important part of the development of new optimization, analysis, and parallelization techniques. Without some sort of quantitative measurement, we cannot really evaluate how well the new techniques work. In order to build up a comprehensive benchmark suite, each person will provide one program that can be incorporated into the suite.

This year, you will be using MATLAB and AspectMatlab. You learned about MATLAB in Assignment #0, and you will be given an introduction to AspectMatlab in class. The web page for AspectMatlab is `http://www.sable.mcgill.ca/mclab/projects/aspect-matlab`. If you get stuck using the tools, please consult with the TAs and share your expertise with other class mates.

When choosing your benchmark, consider the following:

- The benchmark should compute something interesting. Real applications from different branches of Science and/or Engineering would be excellent. Applications solving toy CS problems are **not** interesting.
- You can write your own benchmark or you can use code that someone else has written, but we need to have permission for all of us to use it as a benchmark. It is even better if we are allowed to distribute it as part of our Matlab benchmark set. Something with an open source license is ideal.
- The benchmark should not require proprietary libraries. We need to be able to access and optimize all of the code.
- The benchmark should spend most of its time computing (and not doing I/O).

- The benchmark should not use dynamic features that make the program hard to analyze. For example, it should not make use of functions like `evalin`.
- The benchmark should have deterministic output that can be checked for correctness. We need to be able to run optimized versions of the program and automatically check that the output matches the output of the original unoptimized version. The benchmark should not have a GUI interface that requires human interaction to run, should not produce graphical output, nor should it require any terminal input (we need to be able to run the benchmarks from a script). If you need external files for input (i.e. if you compute on some large data set), please use ASCII files which contain one array each. The benchmark should then use the MATLAB command `load`. Do not use .mat files, because it may make programs hard to analyze.
- Your benchmark needs to have different inputs that allow it to run for different amounts of time. Provide at least two inputs to correspond to a short run of (1-10) seconds, and a long run of >100 seconds.

We created a template for MATLAB benchmarks that satisfies most of these criterias, except it does not compute something interesting:

`https://github.com/Sable/matlab-implementation-template`

You may use it as a starting point for organizing your benchmark code.

Recommended: following the Wu-Wei conventions to leverage existing tools and facilitate reuse

Moreover, the previous template is compatible with the Wu-Wei Benchmarking Toolkit that you were invited to try in assignment #0. By modifying the template to add your code and keeping to the Wu-Wei conventions, you will be able to use the tools to automate some of the manipulations and make your life easier in addition to allowing other students to reuse your benchmark for their course project (and inversely). If the benchmark is sufficiently good, we might even include it into one of our suite and give you credit for the work!

Although the toolkit has been used for two publications and a Master thesis and has proven quite a time-saver, there may still be limitations or bugs that make it unsuitable to the kind of benchmark you would want to choose or requires you to read some extra documentation. Since it is the first year we try it with COMP 621 students and we are calibrating the workload required, it is optional. However, we will be very impressed if the benchmark you hand-in is compatible with the toolkit and therefore easy to reuse.

If you decide to use the toolkit, follow the Wu-Wei specific instructions, indicated with the *-ww* suffix. If you are part of the Sable lab, it is **highly** recommended that you use the tools as they will probably become handy as part of your research project. In any case if you get stuck with the tools more than 20 min. after reading the relevant sections in the handbook (`https://github.com/Sable/wu-wei-handbook/`), preferably write your questions in the

Gitter chat room `https://gitter.im/Sable/wu-wei-benchmarking-toolkit` or otherwise send an email to the TA at `erick.lavoie@mail.mcgill.ca`.

Where to look for a benchmark

There are many places to look. Here are a few possibilities:

- a research group at McGill using Matlab.
- a program you have written in another language, translated to Matlab (this is interesting, because you can compare the performance of the original implementation to your Matlab implementation).
- `http://www.mathworks.com/matlabcentral/fileexchange/`
- `http://web.mst.edu/~gosavia/mrrl\_website.html`
- `https://github.com/trending?l=matlab`

If you plan to use Wu-Wei, it is much easier if the input data is generated from a random number generator before being passed to the computation kernel, as done in the MATLAB template provided. The convention for reading input files are still being determined.

Steps you must do

(a) Give a brief description of your benchmark which highlights the important or interesting characteristics of the program. Describe it both from the scientist/engineering point of view (why this is an interesting/important problem) and from the compiler optimizer point of view (what features make it interesting to optimize).

(a-ww) Do step (a). Then, read the handbook "Overview" `https://github.com/Sable/wu-wei-handbook` and then the "Create a new implementation" guide `https://github.com/Sable/wu-wei-handbook/blob/master/create-new-implementation.md`.

Once you are done, modify the benchmark template and the Matlab implementation template to add your source code and make sure they work with the Wu-Wei tools. `wu list` should list your benchmark. Then install the required compiler and execution environments with `wu install https://github.com/Sable/COMP621-a1.git`. Test your benchmark with `wu run matlab-2016a-jit -v` to test on the latest version of MATLAB. Then test your benchmark with all the versions of MATLAB and Octave with `wu run -v`. Make sure it runs to completion and you see no errors.

(b) Describe how to run the benchmark, and describe the required formats for inputs and which different inputs you have provided.

(b-ww) You can skip this part, the Wu-Wei conventions are doing it for you (Yay!).

(c) Run your benchmark using a Mathworks production version of MATLAB. Run the benchmark 10 times, then report the min, max, average and standard deviation of the 10 runs. You will probably want to write a script to do this task.

Report the version of MATLAB that you used.

Report the architecture and OS on which you did the experiments.

(c-ww) Run the experiments 10 times by typing `wu run -n 10`. Consult the results with `wu report`. Report the min, max, average, and standard-deviation that are computed automatically for you (Yay!).

Report the information that is given you by `wu platform` it should include the OS and architecture and some additional information. The same information is present in the 'runs/latest/run.json' which also include the performance measurements you made.

(d) Define an aspect using AspectMatlab which can be used to count or profile some feature of your benchmark. This could be a very simple aspect which counts the total number of array accesses and/or the total number of function calls. However, it could also be something more interesting. Give the source code for your aspect and explain its purpose.

(e) Compile your original benchmark, along with your aspect using the AspectMatlab compiler. Report on any problems which could be improved by better static checks or better error messages.

(e-ww) Install the AspectMatlab compiler with `wu install https://github.com/Sable/aspect-matlab-compiler.git`. Test that the default aspect used by the compiler correctly compiles your benchmark with `wu build aspect-matlab -v`. Replace the default aspect with your own by:

(1) copying your file into `compilers/aspect-matlab` directory in your Wu-Wei repository;

(2) modifying the `compiler.json` file to replace the `examples/grow.m` file with the path to your file. Test the compilation of your benchmark with the newer aspect with `wu build aspect-matlab -v`.

(f) Run the compiled/woven Matlab code using Mathworks MATLAB and report on runtimes as in (c). You may have problems executing the compiled/woven code on newer versions of MATLAB so use the 2013a (SOCS: `/usr/local/pkgs/matlab/R2013a/bin/matlab`) and 2014b (SOCS: `/usr/local/pkgs/matlab/R2014b/bin/matlab`) versions for your tests.

(f-ww) Run the compiled/woven code on these two versions by doing `wu run aspect-matlab matlab-2013a-jit matlab-2014b-jit -v`. Compare the results with the previous

ones with `wu report`. To report the profiling performed with the aspect, you have at least two options:

(1) Modify the 'runner.m' script to print the information before displaying the timing results. Be careful that your runner still works when the aspect is not woven. The printing should therefore be conditional on the presence of the global object that holds the profiling information;

(2) Start MATLAB from the build directory obtained when doing `wu build -v`. Manually call the runner from the interpreter with the proper parameter(s). Manually print the profiling information;

(3) Anything else you may think of.

(g) Discuss the results of (f). Did introducing your aspect slow down the program substantially? If so, why? Did your aspect provide you with interesting profiling results. If so, what?

(h) Starting with the original benchmark code, profile the execution using the Matlab profile function as documented at `http://www.mathworks.com/help/techdoc/ref/profile.html`, or the IDE profiler as documented `http://blogs.mathworks.com/community/2010/02/01/speeding-up-your-program-through-profiling/`.

Summarize your profile results. Based on this profile discuss which are the most important parts of the program to optimize?

(h-ww) There is no support in Wu-Wei at the moment. You may simply start MATLAB from the benchmark implementation directory and call the runner function directly after activating the profiler.

(i) Based on your observations from (h), find some part of your program that you can optimize by hand (i.e. rewrite the Matlab code for that part to be faster). Perform the rewriting and explain why you think it will improve performance. Using your hand-optimized code, rerun your timings for Mathworks MATLAB and report on the results and the speedup/slowdown as compared to the unoptimized version. Discuss your results.

(i-ww) Follow the instructions in the guide `https://github.com/Sable/wu-wei-handbook/blob/master/create-new-implementation.md` (first case) to add your optimized implementation to your benchmark. Call your optimized version 'matlab-optimized'.

(j-ww) Give feedback on your experience with the tools, both positive and negative. What was easy? What was complicated/hard to understand? Which step(s) took the most time?

What to hand in

- A file called `yourLastName.tar.gz` file containing a directory called benchmarkName/. Of course, replace yourLastName with your real last name!

  Inside this directory you should have a sub-directory called src/ which contains your source code (the original version), all the inputs, all the outputs, a README that describes how to compile and run your benchmark. Also include a sub-directory called aspect/ that contains the source code for your aspect, also with a README describing your aspect. Finally, include a sub-directory called improved/ that contains the source code of your hand-optimized benchmark, along with a README summarizing the differences between this version and the orig/ version.

  Thus, your file should be something like `hendren.tar.gz` and when unzipped it should create a directory structure like:

```
/fibonacci
    /src
       README
       <otherfiles>
    /aspect
       README
       <otherfiles>
    /improved
       README
       <otherfiles>
```

  If you used Wu-Wei you do not have to follow the previous conventions. Simply do the following steps:

  - Remove unnecessary files:
    - `wu build --clean`
    - `rm -rf compilers/aspect-matlab/AspectMatlab`
  - Create an archive with your entire Wu-Wei repository called `yourLastName.tar.gz`. It will contain both the original and optimized versions of your benchmark, the aspect ready to use, and your timing results including the configuration that produced them without any extra step on your part (Yay!).

  Send a link to this file, or as an attachment, to `erick.lavoie@mail.mcgill.ca`. Indicate on the title of the message `cs621 Matlab Benchmark`. Please make sure to send *just one* copy but if you must update what you've already handed in, indicate this in the subject line with the word **UPDATE**.

- A hardcopy of your answers to all parts of this question. Please try to print in a paper-saving fashion. (You do not need to print your benchmark code)

Question 2: *Putting analysis to practice*

Assume the following simplified C-like language which also has some Matlab-like features.

```
<stmt_seq>    ::= <empty_stmt> | <stmt> <stmt_seq>

<stmt>        ::= <basic_stmt> ;  |
                  if ( <cond_expr> ) <stmt> else <stmt>  |
                  while ( <cond_expr> ) do <stmt>  |
                  do <stmt> while ( <cond_expr> ) ; |
                  { <stmt_seq> }

<basic_stmt> ::=  <id> = <id_const> <bin_op> <id_const> |
                  <id> = <id_const>          |
                  <id> = read_int()          |
                  <id> = read_double()       |
                  <id> = read_array( <id_const>, <id_const> ) |
                  <id> = <id_const>  |
                  int( <int_const> ) |
                  sum( <id> ) |
                  transpose ( <id> ) |
                  write( <id> )      |
                  break

<cond_expr>        ::=  <id_const> <relop> <id_const>

<id_const>    ::=  <id> | <int_const> | <double_const> | <array_const>

<bin_op>      ::=  + | * | - | /  | mod

<rel_op>      ::=  < | > | <= | >= | == | !=
```

This language has three types of variables, scalar integers, scalar doubles and two-dimensional arrays of doubles.

Scalar integers are created via a call to an integer creator, (i.e. `int(3)`), or via `read_int`.

Scalar doubles are created via a real constant (i.e. 3, 3.0 or 3e10), or via `read_double`. Note that `3` denotes the same double value as `3.0`.

Arrays of doubles are created via the array constant (i.e. `[3.0, 4.0; 5.0, 6.0]` would create a 2x2 array), or via `read_array`.

The `rel_op` operators must get two scalar arguments, and the result is always the integer 1 for *true* or the integer 0 *false*, these values are used to determine control flow. The `bin_op` operators can have arguments of any of three data types.

The typing rules are dynamic (i.e. checked at run-time) and are summarized as follows:

```
int bin_op int -> int
int bin_op double -> int
double bin_op int -> int
double bin_op double -> double
array bin_op array -> array
sum(array) -> double
transpose(array) -> array
int rel_op int -> int
double rel_op double -> int
```

Any other combination of argument types gives a runtime error.

Consider the following example program.

```
 1:  n = read_int();
 2:  m = read_int();
 3:  x = read_array(m,n);
 4:  y = read_array(n,m)
 5:  if (n > m)
 6:    { s = 0;
 7::      do {
 8:        t1 = x * y;
 9:        t2 = sum(t1);
10:        t3 = s + t2;
11:        n = n - 1;
12:      } while (n > 0);
13:     write(s);
14:    }
15:   else
16:    { p = 1;
17:      i = 0;
18:      x_tr = transpose(x);
19:      while (i < n)
20:        { t1 = sum(x);
21:          x = x + t1;
22:          t2 = sum(x_tr);
22:          if (t2 < 0)
24:            break; % breaks out of closest enclosing loop
```

```
25:           p = t1 * t2;
26:           i = i + 1;
27:         }
28:       write(p);
29:     }
30: write(t1);
31: write(n);
```

(a) Draw a typical control flow graph (CFG) that could be used to represent the entire program fragment, putting the statements in basic blocks where possible. Function calls do not break basic blocks.

(b) Indicate the extended basic blocks for the CFG you created in part (a).

(c) Draw the Dominator Tree for the CFG from part (a). Identify the loops, explaining how how found them.

(d) Assuming that we can represent a definition by a pair *(varname,lineno)*, (i.e. the definition at line 4 would be (y,4)), what are the set of definitions that **may** reach the input of lines 5, 7, 8, 12, 16, 18, 24, 28 and 30 in the example program? As an example, the set of definitions reaching the input of line 2 is { *(n,1)* } .

(e) Based on reaching definitions, what optimizaton(s) could be done on the example program?

(f) What variables are live at the input of lines 30, 28, 26, 22, 18, 16, 13, 10, 7, 5 and 1 in the example program?

(g) We can define a variable **x** to be *super live* at program point p if **x** will be used (before being redefined) and all paths from p to the end. Give the definition of *super live* analysis for the example language, using the steps for defining an analysis you learned in class.

(h) Give a small program fragment which contains the statement `x = y` and on some path `y` is integer, whereas on another path `y` is double. In this case `x` will sometimes be integer, and sometimes `double`.

(i) Give a small program fragment containing a loop which contains the statement `x = transpose(y)`, where `y` has an array type on one path and integer type on another path. In this case there is at least one path that leads to a run-time type error.

(j) Define a static analysis which approximates for each statement, the set of possible type each variable could have on input, and on output of the statement. Show the results of your analysis on the two examples from (g) and (h).

(k) Describe two potential uses of the analysis you defined in (i).

What to hand in

- A hardcopy of your answers to all parts of this question.